Contents lists available at ScienceDirect
Information and Computation
www.elsevier.com/locate/yinco

From Jinja bytecode to term rewriting: A complexity reflecting transformation

Georg Moser, Michael Schaper*

Institute of Computer Science, University of Innsbruck, Austria

ARTICLE INFO

Article history:
Received 1 July 2015
Available online xxxx

Keywords:
Program transformation
Term rewriting
Termination and resource analysis
Automation

ABSTRACT

In this paper we show how the runtime complexity of *imperative* programs can be analysed fully automatically by a transformation to *term rewrite systems*, the complexity of which can then be automatically verified by existing complexity tools. We restrict to well-formed *Jinja bytecode* programs that only make use of non-recursive methods. The analysis can handle programs with cyclic data only if the termination behaviour is independent thereof.
We exploit a *term-based* abstraction of programs within the abstract interpretation framework. The proposed transformation encompasses two stages. For the first stage we perform a combined control and data flow analysis by evaluating program states symbolically, which is shown to yield a finite representation of all execution paths of the given program through a graph, dubbed *computation graph*. In the second stage we encode the (finite) computation graph as a term rewrite system. This is done while carefully analysing complexity preservation and reflection of the employed transformations such that the complexity of the obtained term rewrite system reflects on the complexity of the initial program. Finally, we show how the approach can be automated and provide ample experimental evidence of the advantages of the proposed analysis.

1. Introduction

In recent years research on complexity analysis of first-order term rewrite systems has matured and a number of noteworthy results could be established. First we give a quantitative assessment based on the annual competition of complexity analysers within TERMCOMP.[1] With respect to last year's run of TERMCOMP, we see a success rate of 42% (60%) in the category *Runtime Complexity – Innermost Rewriting*. This benchmark roughly represents runtime complexity analysis of functional programs evaluated in an eager fashion and success here means that a polynomial upper (lower) bound could be established fully automatically. It is known that a considerable amount of examples of the benchmark does not exhibit polynomial runtime complexity or even termination. With respect to a qualitative assessment we want to mention (i) efforts to suit existing termination techniques to complexity analysis [1–4], (ii) adaptions of the dependency pair method to complexity [5–7], (iii) the ongoing quest to incorporate compositionality [8,9], and (iv) very recent work on worst case lower bounds [10]. ([11] provides a survey on methods of complexity analysis of term rewrite systems.)

A natural idea is to exploit existing complexity analysers in the context of (fully automated) resource analysis of programs. Successful examples being for example the development of complexity analysis tools for logic or (higher-order)

* Corresponding author at: Innstrasse 42, 6020 Innsbruck, Austria.
E-mail addresses: georg.moser@uibk.ac.at (G. Moser), michael.schaper@uibk.ac.at (M. Schaper).

[1] http://www.termination-portal.org/wiki/Termination_Competition.

https://doi.org/10.1016/j.ic.2018.05.007
0890-5401/© 2018 Elsevier Inc. All rights reserved.

functional programs [12,13]. In this paper we show how the runtime complexity of imperative programs can be analysed fully automatically by a transformation to term rewrite systems, the complexity of which can then be automatically verified by existing complexity tools. More precisely, we study complexity preservation and reflection of transformations from *Jinja bytecode (JBC) programs* to *constraint term rewrite systems*. Jinja is a Java-like language that exhibits the core features of Java [14]. Its semantics is clearly defined and machine checked in the theorem prover Isabelle/HOL [15]. JBC programs run on the Jinja virtual machine (JVM). In our analysis we focus on non-recursive programs. The analysis can handle programs with cyclic data only if the termination behaviour is independent thereof. We summarise the main contributions of this paper.

– We exploit a *term-based* abstraction of JBC programs within the abstract interpretation framework [16] (Lemma 11). The proposed transformation encompasses two stages.
– In the first stage we employ our idea of term-based abstraction to obtain a novel representation of abstractions of JVM states (Definition 11). We perform a combined control flow and data flow analysis by *symbolically evaluating* JVM states, thus obtaining a finite representation of all execution paths of a JBC program P through a graph, the *computation graph* (Theorem 2).
– In the second stage we encode the (finite) computation graph as *constraint term rewrite system*. These rewrite systems allow the formulation of conditions C over a theory T, such that a rule can only be used if the condition C is satisfied in T.
– We prove that the transformation of P to the constraint rewrite system $\mathcal{R}$ is *complexity reflecting*, that is, the runtime complexity function with respect to P is bounded by the runtime complexity function with respect to $\mathcal{R}$ (Corollary 3). Moreover, we obtain that the proposed transformation is also *non-termination preserving* (Corollary 4). We emphasise that our notion of complexity reflecting abstraction is carefully crafted such that we obtain a constant-factor size overhead in the simulation of the bytecode relation as a rewrite relation.
– Finally, we establish automatability of the transformation by providing a prototype implementation. Our prototype is already capable of yielding automated resource analysis of challenging examples from the literature.

We emphasise that our approach is *total* (Corollary 1). That is, the transformation can be applied to any well-formed, non-recursive JBC program and the computation graph is guaranteed to be finite. The constraints in the obtained rewrite systems are employed to express relations on program variables. In our actual use, we fix the underlying theory to Presburger arithmetic. However, the approach extends to any decidable theory that allows reasoning over the program states. With respect to automation, we show how to combine our proposed term-based abstraction with external shape analyses, as presented for example in [17–19].

Quite principally the established transformation allows for the direct use of rewriting based runtime complexity analysis for the resource analysis of JBC programs. However, currently existing tools for complexity analysis do not (yet) extend to *constraint* term rewrite systems. In order to test the effectivity of the approach, our prototype implements techniques for assessing the runtime complexity of constraint term rewrite systems. Furthermore we present (complexity reflecting) transformations from constraint rewrite systems to (standard) term rewrite systems and integer transitions systems. The systems obtained by these transformations thus can make use of well-known techniques from the literature and existing tools. The prototype is integrated into the general framework of T_CT [20,21].[2]

Structure This paper is structured as follows. In Sections 2 and 3 we fix some basic notions to be used in the sequel as well as the definition of complexity reflecting abstractions. Furthermore, we give an overview over the Jinja programming language. Our notion of abstract states is presented in Section 4. We prove correctness of the abstraction in Section 5, while computation graphs are proposed in Section 6. Section 7 introduces constraint term rewrite systems, fixes the studied theory to Presburger arithmetic, and presents the transformation from computation graphs to rewrite systems. In Section 8 we show how to automate the transformation and give insights about the prototype implementation. Related work is presented in Section 9. Finally, in Section 10 we conclude.

2. Preliminaries

We assume basic familiarity with term rewriting and the Java programming language. In what follows we fix some basic notations as well as the definition of complexity reflecting abstractions.

Let f be a mapping from A to B, denoted $f : A \to B$, then $\mathsf{dom}(f) = \{x \mid f(x) \in B\}$ and $\mathsf{rg}(f) = \{f(x) \mid x \in A\}$. Let $a \in \mathsf{dom}(f)$. We define:

$$f\{a \mapsto v\}(x) := \begin{cases} v & \text{if } x = a \\ f(x) & \text{otherwise .} \end{cases}$$

[2] http://cl-informatik.uibk.ac.at/software/tct.